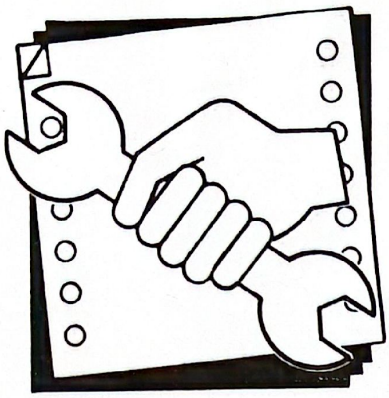
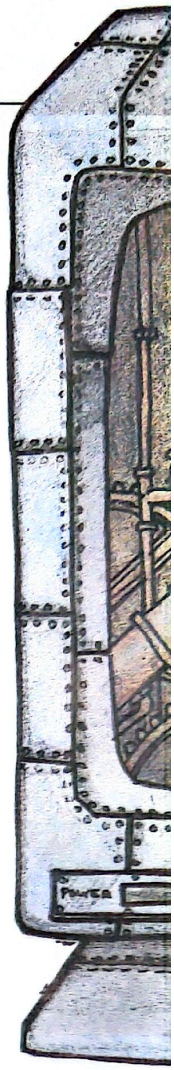
APC
MACHINE CODE



Getting into machine code



Don Thomasson begins his series which will provide an introduction to machine code for users of the PCW8256 and PCW8512 machines. This is your chance to learn the art of talking directly to your machine.

Many computer users are content to rely on Basic for their initial forays into programming, but after a while they begin to feel a desire to spread their wings a little and experiment with machine code, which they understand to be faster and more flexible. Those characteristics, however, are only part of the story.

In its unobtrusive way, Basic gives a programmer a lot of help. Once a variable has been given a name, its value can be found by repeating the name. There is no need to know where the value is stored.

In machine code, the programmer has to look after that little chore himself. If the value is in floating point form the Basic interpreter will carry out the complex calculations automatically, but there is no corresponding help in machine code.

Machine code is faster largely because it does not provide these and similar aids, but it should be said that the difference in working speed is not as great as it was a few years ago, Basic interpreters having been made much faster.

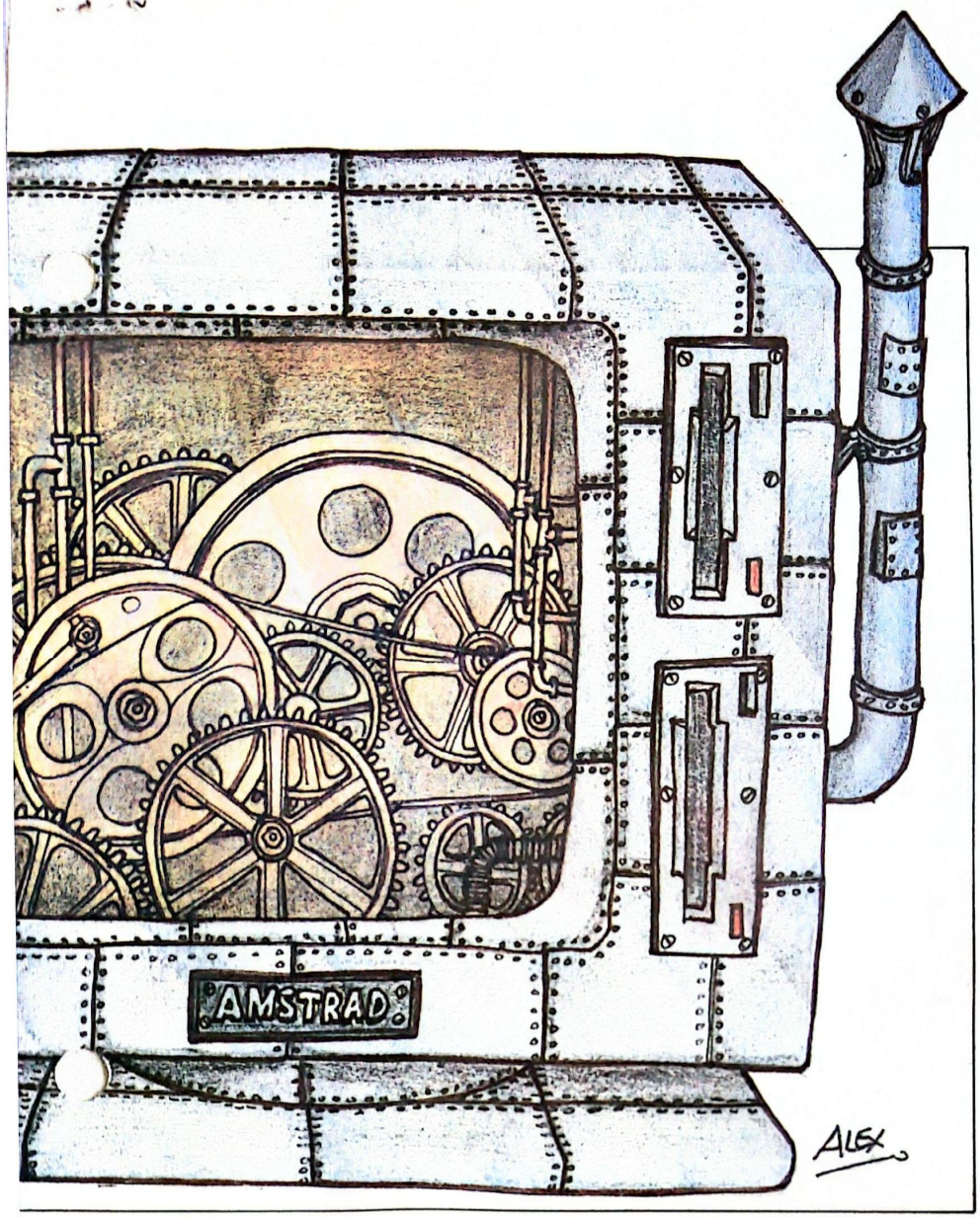
So why bother with machine code? Would it not be better to try Pascal, say? The answer is that all working programs are ultimately in machine code form, the only language that the computer understands. Situations often arise in which a program does not do what it should, but a knowledge of machine code will allow the fault to be corrected.

Those who are ignorant of machine code are therefore at a disadvantage. There is a clear parallel with a person who drives a car without having any idea of what happens in the engine and gearbox. If anything goes wrong, the driver can only yell for help, where a more knowledgeable one might solve the problem with ease.

Basic concepts

Computer programming is a process by which you tell a computer what you want it to do. You give it instructions. Each instruction begins with an operation code, usually called op-code. This may be followed by one or more qualifying codes, defining a store address – a pigeonhole in the computer's memory – or a numeric value.

The op-code and any qualifiers are fetched from store, the required action is executed, and the next instruction is then fetched. A special register called the program counter is used to identify the locations from which the instruction is to



be taken.

This register is normally incremented as each 8-bit byte is read but some instructions, called jumps, can reset the register so that the next instruction is read from another part of the program.

The op-codes are merely binary patterns, having no numeric significance, and it is convenient to use hexadecimal notation to represent them in a neat way. Some writers seem to believe that their readers are insufficiently intelligent to deal with this form of numeric representation, but there is nothing very difficult about it.

The key idea is that any possible arrangement of four binary bits can be represented by a single symbol:

0000 0	0100 4	1000 8	1100 C
0001 1	0101 5	1001 9	1101 D
0010 2	0110 6	1010 A	1110 E
0011 3	0111 7	1011 B	1111 F

For example, the binary number 11001101 can be represented by CD, which is much neater and easier to remember.

Conversion between decimal and binary or hexadecimal is not quite as simple, but the computer itself helps here. There is one snag. The decimal number 1000 is represented in hexadecimal by 3E8, or more usually by 03E8, recognising that a 16-bit number is involved. In a qualifier, however, the halves of the number are exchanged. To load a value of decimal 1000 into the HL register, you need the three bytes 21 E8 03. 21 is the op code, E8 and 03 are the qualifiers. This reversal also applies to address qualifiers.

Necessary data

In order to create a program in machine code you need knowledge of certain essential facts.

You need to know the structure of the central processor and how the registers are arranged. Then you need to know the op-codes, and what they do.

Very soon you will need to know how to make use of the main operating system of the computer to access peripheral devices, such as the display and keyboard. You will also need to know something about the memory map.

These characteristics are sometimes called the environment in which your program has to run. In the present context, we will begin by working from Basic, for the sake of simplicity, but later on we will introduce the CP/M environment, which allows machine code programs to be constructed with greater freedom.

To throw all this information at you en bloc would be merely boring, so we will conclude this section by talking about the way you should begin to write a program.

Top down

The academically correct way to write a program is to create a specification defining the objectives in broad terms, then the requirements are split into functional modules, each specified in adequate detail. From this should emerge definitions of the actual sections of machine code. The number of specification levels will depend on the size and complexity of the requirements, but the principles remain unaltered.

As an illustration, consider a program for handling a database. The top specification needs to define the size and structure of the database blocks and the operations that are to be carried out on them. At the next level down, the first need is for a master control which will allow a required function to be accessed. This might be sketched thus:

a. Display menu of available options
b. Wait for keyboard input
c. Access routine specified by input
d. Go to a

Item *a* will be used elsewhere, and a special routine might be provided, of the form:

e. Copy contents of memory location M to register R
f. If R holds the termination code then exit
g. Display character for code in R
h. Increment M
i. Go to e.

What is wanted here is a routine that can be called with M defined as the start of a text string held in memory. It should be a subroutine, which means a routine that can be called from any point in the program and will return to that point when it has finished execution. We will talk about that later.

Item *g* will need to use the main operating system, but the other items will be single instructions.

The terminating code at the end of the text is commonly zero, but other codes are sometimes used. CP/M, for example, uses Ascii $.

Going back to the main routine one level up, item *b* will need the main operating system. Item *c* requires that the input code should be recognised, and the appropriate routine entered. That looks simple enough, but needs due thought.

● *So far we have done no more than lay the groundwork, but it is essential to get it right. Next month, we will begin to examine the processor structure and the instruction set, beginning to write some simple routines.*

APC